Appears in *Automating Software Design*, AAAI Press, M. Lowry & R. McCartney, 1991, pp. 65-91

Using Evolution Transformations to Construct Specifications

W Lewis Johnson and Martin S. Feather

The Knowledge-based Software Assistant, as proposed in Green et al. (1986), was conceived as an integrated knowledge-based system to support all aspects of the software life cycle. Such an assistant would support specification-based software development: Programs would be written in an executable specification language from which efficient implementations would mechanically be derived. A number of systems have since been developed, each providing assistance for individual software activities. This chapter describes research conducted in the course of developing two of these systems. The first, the Knowledge-based Specification Assistant (KBSA Project 1988; Johnson 1988), was specifically aimed at supporting the evolutionary development of specifications. The second project, ARIES (acquisition of requirements and incremental evolution into specifications), is currently under way. It provides integrated support for both requirement analysis and specification development. ARIES is jointly being developed with Lockheed Sanders.

The original project report anticipated that specifications would evolve but did not describe the mechanism for such evolution. In part as a result of the work on the Specification Assistant, the current vision of an ultimate Knowledge-based Software Assistant embraces the notion of a formalized specification development process (Elefante 1989). In our approach, a description of the system to be built is created in a machine-processible form from the early stages of a software development project and is gradually refined and evolved to produce a formal specification together with supporting documentation. During this process, changes to requirements and specifications frequently occur and must be supported and managed. This chapter presents the mechanism that we have developed to support the process called evolution transformations. The chapter describes how evolution transformations can be employed in developing specifications, and compares this approach to other incremental specification techniques. We provide a detailed description of our representation of transformations and of our mechanisms for retrieving and applying them. Our current efforts at reorganizing the transformation library into basic operators and macrooperators are summarized. The chapter concludes with a discussion of how evolution transformation technology can be applied to other software development tasks.

Evolution Transformation

During the specification development process, a system description undergoes well-defined semantic changes (Goldman 1983). New details are added, revisions are made to resolve conflicts between definitions, and high-level requirements on overall behavior are transformed into requirements on the behavior of individual system components. Evolution of the system description continues as a system is maintained. To support evolution, we have constructed a library of transformations for modifying specifications. This library consists primarily of so-called evolution transformations, that is, transformations whose purpose is to elaborate and change specifications in specific ways. Like conventional correctness-preserving transformations (also called meaning-preserving transformations), they can be invoked by the user or by other transformations, and they

are executed by a mechanical transformation system to cause changes to a specification. Correctness-preserving transformations are generally applied to derive efficient implementations from specifications, keeping the meaning of the specification unchanged; in contrast, our evolution transformations deliberately change the meaning of specifications. We do, in fact, include some meaning-preserving transformations in our library, but instead of deriving efficient implementations, their purpose is to reorder specifications (for better presentations), rewrite specifications into equivalent forms using different language constructs, eliminate redundancies, or make explicit some otherwise implicit specification features. Some of these transformations can also appear in a transformational implementation system, to be used to replace high-level specification constructs with low-level implementation ones.

Our evolution transformations perform semantic changes, such as revising type hierarchy defined in a specification, changing data-flow and control-flow paths, and introducing processes to satisfy requirements. A single transformation can perform a number of individual changes to a specification; for example, if a definition is changed, all references to this definition throughout the specification can be changed in a corresponding manner to retain semantic consistency.

Advantages of Transformational Evolution

The evolutionary development of specifications by transformation has several advantages. First, because the transformations mechanically take care of low-level editing details, changes can be succinctly directed and reliably performed. This result is analogous to the advantage of using conventional correctness-preserving transformations for deriving efficient implementations from specifications, cogent arguments for which can be found in Balzer, Goldman, and Wile (1976) and Bauer (1976).

Second, the transformational development of specifications facilitates the separation of concerns during the analysis process. The initial analysis of requirements can focus on modeling the system environment and describing the effects that the intended system will have on this environment. These effects can be stated with little regard for the particular design that will achieve them. As a separate activity, the analyst can then propose a design and transform the requirement statements into constraints on the design. Evolution transformations have two distinct roles in this process: to aid in constructing the initial model of environment and requirements and to transform the requirements into specifications in a way that preserves the meaning of the requirements as much as possible.

Third, the record of transformation steps preserves the history of how high-level requirements are developed into lower-level specifications. This approach provides traceability, which is helpful for understanding the resulting specifications and assuring that all the original requirements have suitably been incorporated.

Fourth, the transformation record can be undone and replayed, permitting the exploration of different specification choices. This approach permits a developer to explore several (relatively) unrelated evolutions to the same specification, with the resulting separation of concerns and its attendant benefits. To do this exploration, the same starting specification is separately evolved into several specifications, and the transformational record is kept in each case; the resulting (multiple) specifications can then be combined by serially replaying all of the transformations on the initial specification (Feather 1989a). The same method also permits multiple developers to independently make changes to a common specification, combining their changes later. Where the separate evolutions are not, in fact, independent, comparison of the transformations can reveal the

interference (Feather 1989b). This development model was extended to the case in which the separate evolutions describe different users' conflicting requirements, and negotiation techniques are utilized to resolve the conflicts (Robinson 1989).

Last, when a specification has been transformed into an efficient implementation and is later to be changed, conducting the change through the use of evolution transformations can facilitate the replay of the original transformational development on the changed specification (Feather 1990).

Developing a Library of Transformations

We began our exploration of evolution transformations by concentrating on two problems, a patient monitoring system and an air traffic control system, and manually worked out development scenarios to discover what transformations were necessary. We then implemented general-purpose versions of these transformations, which could be applied to mechanically achieve these developments. The result of this exploration was a sizable library containing about 100 transformations of a wide variety of types. This library is significantly more extensive than similar libraries developed by Balzer (1985) and Fickas (1987). In addition, although other researchers have studied evolution steps similar to those captured by our transformations (Narayanaswamy 1988; Johnson 1989), they have not developed transformations to enact these steps.

The ARIES system expands on this work; we are developing a transformation library that is extensive enough and powerful enough to apply to a wide range of specifications. Two issues have been of particular concern in the current work. First, a method is needed for characterizing the effects of evolution transformations. This method is necessary to ensure the coverage provided by the library (that is, to determine what range of transformations is required in the library for it to support a wide range of analyst activities) and to retrieve from the library (that is, retrieve the appropriate transformation to make the desired specification change). Second, we need to be able to apply transformations to specifications expressed in a variety of notations. Whereas the Specification Assistant operated only on specifications expressed in the specification language GIST (Goldman et aI. 1988), ARIES supports a wide spectrum of other notations, including hypertext, flow diagrams, stale-transition diagrams, and domain-specific notations. We needed a common internal representation capturing the semantics of all these notations. By applying the transformations to the internal representation, the same transformation library can be applied to specifications expressed in a variety of notations.

The solution to both problems required identifying the different semantic dimensions embodied in a specification. The ARIES system internally represents specifications as descriptions along each of these dimensions. These descriptions are relatively independent of any particular notation that might be used to express these semantics. This arrangement provides a means of characterizing the effects of transformations: Each transformation performs specific changes to one or more semantic dimensions of the specification. Adequate library coverage can then be achieved by making sure that all possible changes along each semantic dimension are supported. Notation independence makes it possible for analysts to use the same transformations to edit different notations. When an analyst proposes a change to a particular view of a specification, this change can be along one or more semantic dimensions, which, in turn, can suggest appropriate transformations to apply.

Related Work

Burstall and Goguen (1977) argue that complex specifications should be put together from simple ones and developed their language CLEAR to provide a mathematical foundation for this construction process. They recognize that the construction process itself has structure, employs a number of repeatedly used operations, and is worthy of explicit formalization and support-a position that we agree with.

Goldman (1983) observes that natural language descriptions of complex tasks often incorporate an evolutionary vein: The final description can be viewed as an elaboration of some simpler description, itself the elaboration of a yet simpler description, and so on, back to some description deemed sufficiently simple to be comprehended from a non-evolutionary description. He identifies three dimensions of change between successive descriptions: *structural*, concerning the amount of detail the specification reveals about each individual state of the process; *temporal*, concerning the amount of change between successive states revealed by the specification; and *coverage*, concerning the range of possible behaviors permitted by a specification. We were motivated by these observations about description to try to apply such an evolutionary approach to the construction of specifications.

Fickas (1986) suggests the application of an AI problem-solving approach to specification construction. Fundamental to his approach is the notion that the steps of the construction process can be viewed as the primitive operations of a more general problem-solving process and, hence, are ultimately mechanizable. Continuing work in this direction is reported in Robinson (1989) and Anderson and Fickas (1989). Fickas and his colleagues have concentrated on domain-specific goals arising in the course of specification development, whereas our efforts have concentrated on problem-independent goals.

In the Programmer's Apprentice project (Rich, Schrobe, and Waters 1979; Waters 1985), the aim-to build a tool that will act as an intelligent assistant to a skilled programmer-focuses on a different part of the software development activity than our work; however, much of what they have found has relevance to our enterprise. In their approach, programs are constructed by combining algorithmic fragments stored in a library. These algorithmic fragments are expressed using a sophisticated plan representation, with the resulting benefit of being readily combinable and identifiable. Their more recent project on supporting requirements acquisition, the Requirements Apprentice (Reubenstein and Waters 1989), addresses the early stages of the software development process and includes techniques that are similar to those of the Programmer's Apprentice but that operate on representations of requirements. The use of the Programmer's Apprentice is, thus, centered on the selection of the appropriate fragment and its composition with the growing program, with minor transformations to tailor these introduced fragments. In contrast, our approach centers on the selection of the appropriate evolution transformations and the reformulation of abstract descriptions of system behavior using such transformations. However, the two approaches are closely, related. Many evolution transformations instantiate cliches as part of their function. We are currently explring ways of making these cliches more explicit in our transformation system.

Karen Huff (Huff and Lesser 1987) developed a software process modeling and planning system that is in some ways similar to ours. Her GRAPPLE language for defining planning operators influenced our representation of evolution transformations. Conversely, her metaoperators applying to process plans were influenced by our work on evolution transformations.

Kelly and Nonnenmann's WATSON system (chapter 3) constructs formal specifications of telephone system behavior from informal scenarios expressed in natural language. Their system formalizes the scenarios and then attempts to incrementally generalize the scenarios to produce a finite-state machine. Their system is able to assume significant initiative in the formalization

process because the domain of interest, namely, telephony, is highly constrained and because the programs being specified, call-control features, are re1atively small. Our Work is concerned with larger, less constrained design problems where greater analyst involvement is needed. It is also aimed toward the construction of specific behaviors that start from general requirements. Nevertheless, we have recognized for some time that acquisition from scenarios is a useful complement to the work we are doing in highly constrained design situations (Johnson 1986; Benner and Johnson 1989).

The PRISMA project (Nislder, Malbaum, and Schwabe 1989) is also a system for assisting in the construction of specifications from requirements. It has the following main characteristics: First are multiple views of the (emerging) specification, where the views that they explored are data flow diagrams, entity-relationship models, and Petri nets. Second, each view is represented in the same underlying semantic net formalism but represents a different aspect of the specification. This representation is suited to graphical presentation and admits to certain consistency and completeness heuristics whose semantics depend on the view being represented. For example, the lack of an input link has a different interpretation in each diagram. In a data flow diagram it indicates a process-lacking input; in an entity-relationship diagram it indicates an entity with no attributes; and in a Petri net diagram it indicates an event with no preconditions (prior events). Third, heuristics exist to compare the different views of (different aspects of) the same specification and aid in constructing new views or support checking for partial consistency between views. Fourth, errors detected by these heuristics are added to an agenda of tasks requiring resolution along with advice on how to accomplish this resolution. Fifth, a paraphraser produces natural language presentations of many of the kinds of information manipulated by the system (for example, of the requirement information represented in the different views, the agenda of tasks and advice for performing these tasks, the results of the heuristics that detect uses of requirement freedoms).

There is a striking similarity between the approach of the PRISMA project and ours - the use of multiple views, their presentations, and an underlying semantic net formalism. These researchers clearly thought about and developed heuristics to operate on or between views, an aspect that we only recently began to address. Conversely, we provided more support for evolution.

An Example of Transformational Specification Development

Our transformational approach to specification development is not specific to any particular domain. We examined several different domains, including hospital patient monitoring systems and library systems. However, to demonstrate the power and scalability of the approach, we devoted significant effort ~ a particular domain, namely air traffic control. We have been modeling requirements for the Berlin Air Route Traffic Control Center (BARTIC) for air , traffic control in the airspace around Tempelhof Airport in Berlin. We also studied the requirements for U.S. domestic in-route air traffic control systems, that is, those systems responsible for the control of air traffic cruising at the high altitudes reserved for jet aircraft. These requirements are drawn from manuals on pilot and controller procedures (Aviation Supplies 1989; Air Traffic Operations 1989) and the experiences of the current Federal Aviation Administration Advanced Automation Program (Hunt and Zellweger 1987), whose goal is to develop the next generation of air traffic control systems.

In this chapter, we focus on a particular part of the air traffic control problem. One duty of an air traffic control system is to monitor the progress of controlled aircraft and ensure that they adhere to

their planned courses. We examine how the process of monitoring aircraft flights is transformed during specification development.

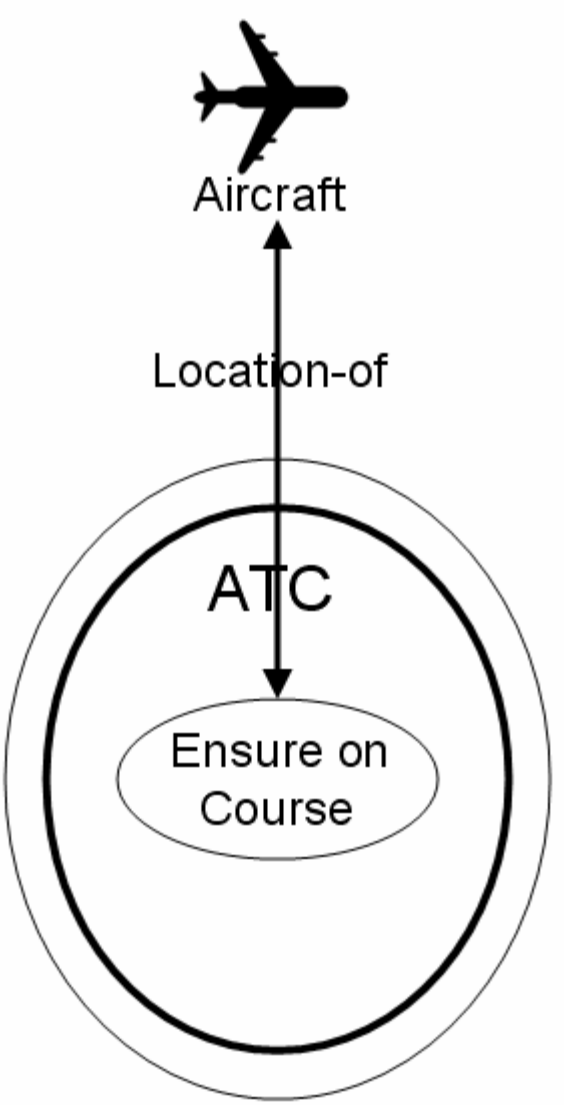

Figure 1. Initial Context Diagram of the Air Traffic Control System

Figure 1 shows an initial view of aircraft course monitoring. We use a context diagram, which shows the interactions between a system and its external environment and the information that flows between them. In these diagrams, ovals denote processes, boxes and miscellaneous icons denote objects, and double circles indicate system boundaries. The diagram distills course monitoring to its essential elements: the interaction between aircraft and the air traffic control system. The air traffic control system has a process called Ensure-On-Course as one of its subfunctions. It examines the location of the aircraft and compares it against the aircraft's expected location. If the two are sufficiently different. the air traffic control system attempts to affect a course change, changing the location of the aircraft.

This abstracted view of the air traffic control system is useful as a basis for stating course monitoring requirements. It is a natural abstraction for the domain, corresponding to the way flight

procedures are commonly described in flight manuals (Aviation Supplies 1989). We do not go into details here about how much the expected location and the actual location are permitted to differ. Instead, we discuss how any such requirements can be transformed into specifications of system function.

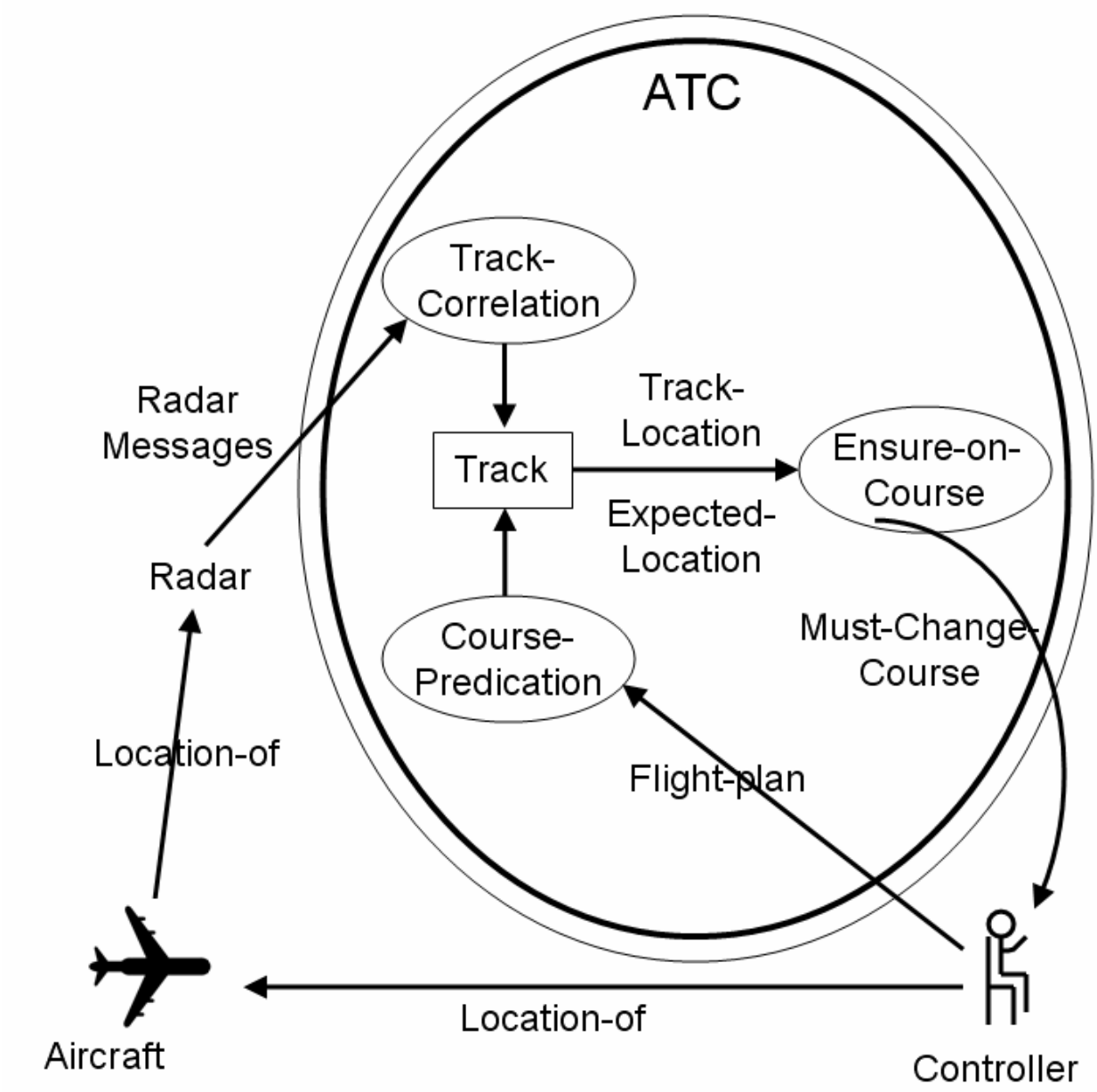

Figure 2. Detailed Context Diagram of the Air Traffic Control System

Figure 2 shows a detailed view of the air traffic control process: More of the agents of the proposed system are introduced, specifically, radars and controllers. The air traffic control system is no longer viewed as a single agent; instead, there are two classes of agents, the air traffic control computer system and the controllers. Determining the locations of the aircraft is performed as follows: The radar observes the aircraft and transmits a set of radar messages, indicating that targets have been observed at particular locations. A Track-Correlation function inputs these radar

messages and processes them to produce a set of tracks. Each track corresponds to a specific aircraft; the locations of the tracks are updated as the aircraft positions change. Meanwhile, expected aircraft locations are computed from the aircraft flight plans, which, in turn, are input by the controllers. The Ensure-On-Course process is modified so that it issues notifications to the controller (by signaling Must-Change-Course for an aircraft); the controller then issues commands to the aircraft over the radio.

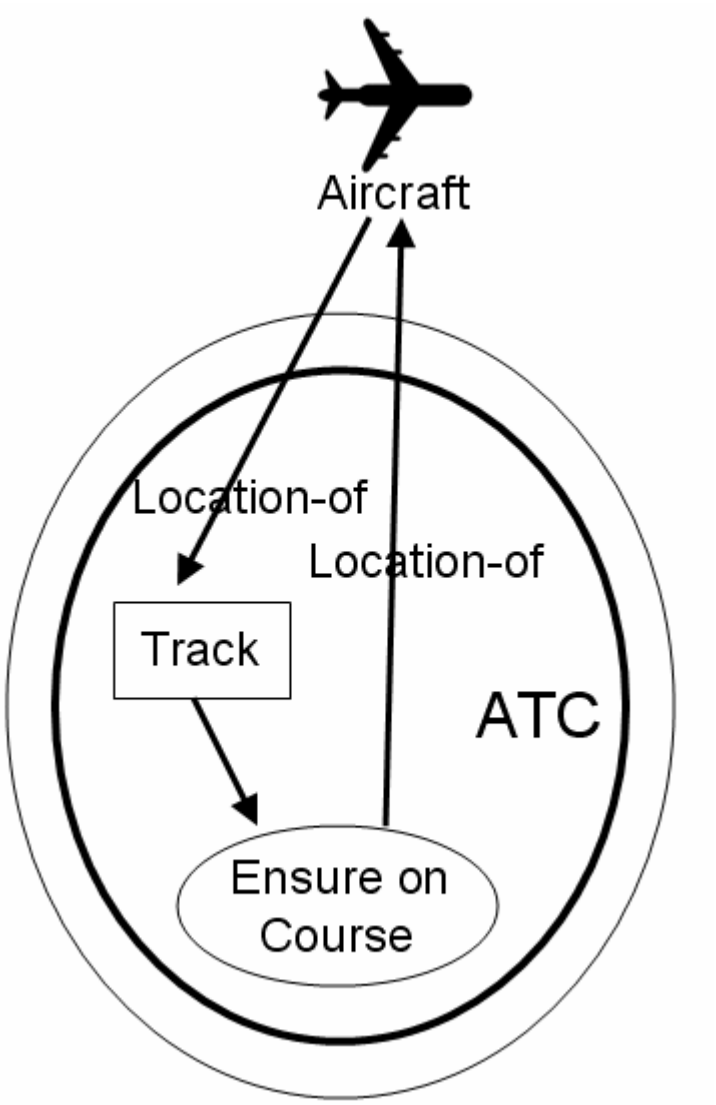

Figure 3. Intermediate Context Diagram of Air Traffic Control System

To get to this detailed level of description, a number of transformations must be performed. Most of the transformations have to do with designing the pattern of data flow through the system. We implemented a number of the evolution transformations necessary to carry out this transformation process. The most important one is called Splice-Data-Accesses. Figure 3 shows the result of applying this transformation to the version in figure 1. It operates as follows: In the initial version, Ensure-On-Course directly accesses aircraft locations. Splice-Data-Accesses is used to introduce a new class of object, called Track, which has a location that matches the aircraft's location. The

Ensure-On-Course process is modified in a corresponding way to refer to the track locations instead of the aircraft locations.

This example is typical of how evolution transformations work. The transformation modifies one aspect of the specification (data flow) and keeps other aspects fixed (for example, the function of Ensure-On-Course). It accomplishes this through systematic changes to the specification. In this case, the transformation scans the definition of Ensure-On-Course looking for references to Location-of; each of these references is replaced with a reference to the Track-Location attribute of tracks.

Completing the derivation of this example requires further application of the following transformations: Splice-Data-Accesses is again applied to introduce the object Radar-Message, which is an intermediate object between Aircraft and Track. Maintain-Invariant-Reactively is invoked to construct processes for continuously updating the radar messages and the tracks. A transformation called Install-Protocol is used to introduce a notification protocol between the Ensure-On-Course process and the controller, so that Ensure-On-Course issues notifications to the controller whenever the location of the aircraft must be changed. A new process called Course-Prediction is added to compute expected locations from flight plans. Through this derivation, the specification is gradually refined into a version in which each system component interacts only with those data and agents that it will be able to interact with in the implemented system. The specification is now ready for detailed design and implementation.

Characterizing Transformations Along Dimensions of Semantic Properties

The fundamental idea underlying our work is the ability to view a specification along a number of different semantic dimensions, for example, a data flow dimension and an entity-relationship dimension. We then characterize our evolution transformations by the effects they induce along each dimension; for example, one transformation might add a new node to the entity-relationship dimension without changing the data flow. Network notations such as entity-relationship diagrams and data-flow diagrams are commonly in describing systems; we take the logical progression of this idea and describe each of our dimensions as a semantic network of nodes and links (relations) connecting these nodes. Based on these descriptions, we characterize the effects of an evolution transformation in terms of generic network-modification operators (for example, add a node; insert a link between two nodes) applied to the various dimensions.

Given the right network abstraction and an appropriate notation for presenting it, transformations with complex effects can be viewed simply and intuitively. However, our approach goes beyond simply providing editors for particular diagrams, as is common in CASE tools. Additionally, we draw a strong distinction between the representation of specifications, such as a semantic network, and the presentation of specifications. This idea was previously introduced in the Knowledge-Based Requirements Assistant and other systems with advanced user interfaces. Because we define our transformations in terms of the representation rather than the presentation, the same transformation can be applied to any presentation that depicts the affected semantic dimensions. Some transformations can simultaneously affect multiple semantic dimensions, resulting in changes to an even broader range of presentations. Thus, for example, Splice-Data-Accesses changes both the information flow of a system (by rerouting data accesses) and the entity-relationship model of the system (by introducing new intermediate objects and attributes). The system's information flow can be viewed using a context diagram or a more conventional data flow diagram; the system's

entity-relationship model can be viewed using an entity-relationship diagram or an inheritance hierarchy diagram.

Background: An Outline of Our Specification Semantics

To understand the semantic dimensions and their effects, we must give an overview of the semantic concerns that we attempt to represent and manipulate. Our goals have been to (1) represent semantic concerns that are commonly recognized as important in requirements engineering and AI, (2) support the translation of commonly used notations into and out of our framework, and (3) support our own research in requirements modeling and design. The result is a semantic framework that supports many common notations.

The basic units of the ARIES system descriptions are types, instances, relations, events, and invariants. These units are grouped with a simple modularization mechanism called *folders*. The treatment of types, instances, and relalations is compatible with most object-oriented approaches to requirements engineering (for example, Hagelstein 1988). However, our entity-relationship model is more general and expressive than most in supporting a wide range of entity-relationship notations. For those readers who are familiar with such systems, the following list summarizes the specific features that our entity-relationship system supports: First, each type can have multiple subtypes and supertypes. Second, each instance can simultaneously belong to any number of types. Third, relations hold among any types of objects; there is no restriction that these types be primitive with respect to any particular machine representation. Fourth, relations need not be binary but can have arbitrary arity. Fifth, relations are fully associative; there is no need for separate relations to record the inverse of a given relation.

System descriptions can describe behavior over time, modeled as a linear sequence of states. Each state is fully described in terms of what instances exist, what relations hold between them, and what events are active.

Events subsume all system processes and external events described as part of a system description. Events have duration, possibly spanning multiple states in a behavior and involving multiple entities of the system. Events can have preconditions, postconditions, and methods consisting of procedural steps. They can explicitly be activated by other events or can spontaneously occur when their preconditions are met. They can have input and output.

Not all interactions with an event must occur through its input and output ports. It is often useful, particularly at the early stages of system specification, to describe events without concern for the specific input and output. For example, the early version of the Ensure-On-Course event described in An Example of Transformational Specification Development directly observed aircraft and modified their locations. An aircraft cannot be considered an input in the conventional sense here. Event declarations whose purpose is to describe activity, rather than specify particular artifacts, tend to have this flavor. Information flow here refers to any transfer of information between agents and their environment. The transformation example in An Example of Transformational Specification Development is aimed at transforming idealized information flows into concrete data flows.

Invariants are predicates that must hold during all states in the system. Invariants are divided into subclasses according to their intended function. Domain axioms are predicates about the environment that are assumed to hold, such as the configuration of airspaces. These invariants will hold regardless of what the system being specified might or might not do. Functional constraints are invariants that involve the system being specified or that are to be guaranteed by the system

being specified. Thus, they are a kind of functional requirement and must explicitly be implemented or respected in the system being specified. An example of such a constraint is the requirement that aircraft not deviate from their designated courses by more than a set amount. Dependency links are established during the design process between such requirements and the events or other specification components (for example, Ensure-On-Course) that are intended to satisfy them.

Folders are used to organize specification information. Each folder contains a set of concept definitions. A folder can inherit from other folders, meaning that concepts within the folder can refer to concepts appearing in the inherited folders. A folder can also import specific concepts from other folders; for example, it can be used to select the correct concept if the inherited folders contain multiple concepts of the same name. It is also possible to give inherited concepts new names In the context of a folder, for example, renaming an inherited concept direction to heading. Folders are the principal mechanism for encapsulation and reuse. ARIES has an extensively populated library of generic and domain-specific folders.

As part of our current research, we are investigating the use of parameterized folders. *Parameterized folders* contain free variables, which must be bound when the folder is used. An example of such a folder is tracker-concepts, which defines concepts related to tracking, such as trajectories, location prediction, and smoothing. This folder contains a free variable, tracked-object-type, which is the type of the object being tracked (for example, aircraft). Such folders are used by instantiating a copy of the folder with the the variables bound, for example, specifying that the value of tracked-object-type is aircraft. The result is the definition of a tracker of aircraft positions. Such parameterized folders are an important mechanism for representing requirement cliches, as in the Requirements Apprentice (Reubenstein and Waters 1989). Many transformations introduce specification constructs having a stereotypic form; Splice-Data-Accesses is one such transformation. The form of the intermediate object created by this transformation can be stored in a folder and instantiated as needed. We expect to make increasing use of such parameterized folders in our transformation library.

Associated with specification components are a variety of attributes, including nonfunctional ones. We do not dwell on these details here (see Harris [1988]). The main conclusion that the reader should draw from this discussion is that most important requirement notations can be captured in this framework, particularly those employing diagrams. Likewise, knowledge representation schemes oriented toward concept modeling are readily accommodated in this scheme as well. The framework was partly implemented in two systems: ARIES and the KBSA Concept Demonstration system (DeBellis 1990). The following translators were implemented to translate several notations into or out of (or both) this ARIES framework (for translation out of ARIES, this function is only possible for those concepts for which a corresponding concept exists in the target): GIST (Balzer et al. 1983), (most ot) the REFINE language (derived from the V language [Reasoning Systems 1986]), (most ot) LOOM (MacGregor 1989), entity-relationship diagrams, concept hierarchies, and ENGLISH (Grove et al. 1971) (but only from ARIES into ENGLISH [Swartout 1982]; we have not pursued natural language understanding as input). Translators for context diagrams, state-transition diagrams, and information flow diagrams are currently under development.

Dimensions of Semantic Properties

In our studies of specification evolution, we have found the following dimensions of semantic properties to be important for characterizing the changes that occur: (1) the modular organization of the specification, that is, which concepts are components of which folders and which folders inherit from which folders; (2) the entity-relationship model defined in the specification, that is, what relations might hold for each type, what attributes it can have, what generalizations and specializations are defined, and what instances are known; (3) information flow links, indicating for each process or event what external information it accesses, what facts about the world it can change, and what values are computed and supplied; (4) control-flow links, indicating what process steps must follow a given process step and what process steps are substeps of a given process step; and (5) state-description links, associating statements and events with preconditions and postconditions that must hold in the states before and after execution, respectively.

Each semantic dimension is modeled using a collection of relations, each representing one aspect of the dimension previously described. Thus, for example, the entity-relationship model is captured using the relations specialization-of, parameter-of, type-of, instance-of, and attribute-of. This model makes distinctions that are missing from many of the notations being supported. Thus, entity-relationship diagrams typically show specialization-of as just another relation in the application's data model. Here, it treated not as part of the application's data model but of ARIES'S language for structuring data models.

This semantic model captures information beyond what conventional notations typically show; however, conventional diagrams can easily be generalized to capture such information. For example, entity-relationship diagrams are generally used only to show relationships among types, whereas our entity-relationship dimension also includes instances. However, entity-relationship style diagrams could also be used to describe instances. The information flow dimension generalizes conventional data flow; it captures the flow of information that is not mediated by conventional message passing. Thus, we can describe air traffic control as monitoring aircraft locations and changing them without implying that the aircraft are somehow sending location messages to the air traffic control system. Still, we can easily generalize conventional data flow diagrams to show such abstract information flow.

Generic Network-Modification Operations

Because we represent each semantic dimension as a semantic network of nodes and relations, we are able to identify a number of generic network-modification operations that apply to any semantic network and, thus, to each semantic dimension. The most primitive network-manipulation operations are insert and remove for adding and deleting links and create and destroy for creating and destroying objects. The meaning of an operation depends on the semantic dimension to which it is applied and the relation being affected; thus, for example, the operation of adding a link in the information flow dimension could mean making a process access information about an external object, whereas the same operation in the entity-relationship dimension could mean making one type become a specialization of another.

In addition to these primitive operations, we identified a number of frequently recurring complex operations:

- Update – Remove a link from one node, and add it to another node.
- Promote (a specialization of update) – if one of the linked nodes is part of an ordered lattice, then update the link so that it connects a higher node in the lattice.

- Demote (the opposite of promote) – move the link to a lower node in the lattice.
- Splice – Remove a link from between two nodes A and B, and reroute the connection through a third node, C, so that A is linked to C, and C is linked to B.
- Split – Replace a node A with two links B and C, linked in some fashion, where B and C divide the attributes of A.
- Join – Replace two nodes A and B with a node C, merging their attributes.

Examples of Dimensions of Semantic Properties and Changes within Them

We sketch some instances of semantic properties that arise in our specification of air traffic control. These examples show how information is actually captured along the different dimensions previously outlined and illustrate the semantic distinctions that are made along each dimension.

Modular Organization: The concepts of mass, direction, mobile object, and location are components of the physical-object folder. The concepts of aircraft, airport, control tower, and so on, are components of the ate-model folder. Three folders are inherited folders of the ate-system folder: (1) atc-model, containing objects and activities common to air traffic control; (2) system, containing definitions of various categories of systems, for example, signal-processing system; and (3) upper-model, a collection of generic concepts for modeling the semantics of natural language defined by the PENMAN project (Bateman 1990). The ate-model folder, in turn, has nine inherited folders, including physical-objects, vehicle, system, and upper-model. The concepts in a folder can be defined in terms of the concepts inherited from other folders, for example, the ate-model's air location is defined in terms of the physical object's location.

Entity-Relationship Model: The specialization relationship is used to express the type hierarchy; for example, aircraft is a specialization of vehicle, which, in turn, is a specialization of mobile-object. Similarly, the instance-of relationship is used to express which types an object belongs to, for example, the bartcc-facility is an instance of the type atc-facility.

Information Flow: As discussed earlier, information flow involves the transfer of information (accesses to, and modifications of, information) between components. For example, the early versions of the Ensure-On-Course event access and modify aircraft locations; hence, both kinds of information flow links, accesses-fact and modifies-fact, hold between Ensure-On-Course and aircraft. Some of these information flows are later transformed into concrete data flows. The data flow relationship expresses the flow of data between components, for example, from the radar process to the Track-Correlation function and from the Track-Correlation function to the Ensure-On-Course process.

Control Flow: There are two kinds of control-flow links: control-substep and control-successor. Control-substep captures the flow of control when an event consists of a series of steps; the relationship holds between the event and its substeps. For example, Track-Correlation has the operation to update an individual track as a substep. Control-successor holds between actions that are in temporal sequence; for example, Ensure-On-Course is activated whenever Track-Correlation updates tracks. A third category of control link, describing causal relationships between events, will need to be included as well, along the lines that Yue (1989) developed for the Specification Assistant.

State Description: Links of this kind are between events and their preconditions and postconditions; for example, a precondition to Ensure-On-Course taking action is that an aircraft be off course, and its postcondition is that the aircraft be back on course (this postcondition is true at least in the early versions of the specification; in later versions, the postcondition is that it has

triggered the activity of notifying the controller, which ultimately causes the aircraft to return to its course).

The meaning of a modification operation will depend on the semantic dimension to which it is applied: In the entity-relationship dimension, to insert a specialization-of link means to assert that one concept is a specialization of another, for example, that the type surveillance-aircraft is a specialization of the type aircraft. In the information flow dimension, to remove an accesses-fact link means to remove accesses to a category of external information from a component; for example, to remove access by atc-system to the aircraft location-of relation. For the specialization links of the entity-relationship dimension, to splice means to assert that some type is intermediary to two other types in the specialization hierarchy; for example, splicing military aircraft between aircraft and surveillance aircraft. In the information flow dimension, to splice means to reroute an information flow between two components through an intermediary; for example, our earlier splicing of track between aircraft and the air traffic control system. In the control-flow dimension, to splice means to reroute a direct control flow between two components through an intermediary; for example, in the early versions of the specification, there would be a direct control flow from an aircraft's Maneuver process to air traffic control Ensure-On-Course process, whereas in later ver1sions, this direct link would have been spliced through the Track-Correlation process.

Transformation Details

We now look in detail at what evolution transformations do and discuss how they operate on the semantic network model of specifications. The discussion centers on a particular transformation, Splice-Data Accesses.

Figure 4 shows offline documentation generated by the ARIES system for the Splice-Data-Accesses transformation. Both ARIES and the Specification Assistant can generate offline documentation of transformations (to be included in manuals and reports) and online help. In the Specification Assistant, the online help provided guidance for the user about what parameters must be supplied to the transformation, what types they should be, and how they should be input. In ARIES, this online help is being integrated into the ARIES diagram editing capability currently under development.

Transformations Operate on the Metamodel

As described in Background: An Outline of Our Specification Semantics, our systems support world modeling in terms of a semantic network of entities and relationships. Events perform actions that create or destroy entities and change relationships. At the same time, transformations are understood as operating on a semantic network of entities and relationships. Thus, the space of possible system descriptions is itself a domain that can be modeled in the ARIES framework. This model of specification objects and relationships is called the ARIES metamodel.

The ARIES metamodel is implemented as a set of types and relations in ARIES. The model is divided into two ARIES folders: the User-Metamodel, consisting of those types and relations that an ARIES user might need to be aware of, and the Lisp-Environment, consisting of those types and relations that are only used internally by ARIES. The user metamodel includes those concepts previously described: types, relations, events, invariants, folders, and so on. Outside evaluations of the Knowledge-Based Requirements Assistant and the Specification Assistant (Abbott 1989)

indicated a need for Knowledge-based Software Assistant systems to have an understanding of their own system model; this approach provides such a model.

```
Splice-Data-Accesses: Transformation
Concept description: Splice a data object into the information-access path from object to an agent or
activity. The transformation can be applied when the agent or activity, called Accessor, accesses
some relation Accessed-Relation, which is an attribute of some object Accessed-Object. The
transformation modifies the definition of Accessor so that it does not access Accessed-Relation
anymore. It performs this modification as follows. It creates a new type Intermediary-Object, with a
new attribute Intermediary-Rel, which correspond to Accessed-Object and Accessed-Relation,
respectively. Every reference to Accessed-Relation in Accessor is replaced with a reference to
Intermediary-Rel. The result is a specification with the same behavior as before but with a different
pattern of information flow.
Note: This transformation creates a new type and new relations. If you instead want to modify existing
types or relations, you might want to use the more general transformation Generalized-Splice.
Input parameters:
  Accessor: Entity:
    Component currently accessing the relation
  Accessed-Relation: Entity: Relation currently being accessed directly
  Accessed-Object: Entity: Object type that Accessed-Relation is an attribute of
  Intermediary-Object-Type-Name: Entity:
    Name of type of intermediary object
  Intermediary-Rel-Name: Entity:
    Name of new object's relation, to be accessed instead of Accessed-Relation
  Correspondence-Rel-Name: Entity:
    Name of new relation mapping old object to new data object
Output parameters:
  Intermediary-Object:
    New object type, named Intermediary-Object- Type-Name
  Intermediary-Rel:
    New object's relation, to be accessed instead of Accessed-Relation
  Correspondence-Rel:
    New relation mapping old object to the intermediary object
Precondition: The Accessor must be an event or type declaration.
Goal: The Accessor does not access the Accessed-Relation.
Main effects:
  An Accesses-Fact relation between Accessor and Accessed-Relation is spliced.
  A Type-Declaration named Intermediary-Object is created.
  A Relation-Declaration named Intermediary-Rel is created.
```

Figure 4. Machine-Generated Documentation for Splice-Data-Accesses

The ARIES system as a whole is implemented in AP5 (Cohen 1989), a set of database programming extensions to Lisp developed at USC/lnformation Sciences Institute. AP5 is the language that specifications are compiled into. That is, to test and simulate specifications, specifications are compiled into a prototype in AP5 and Lisp so that analysts can set up simulation scenarios and run them. In an analogous fashion, the ARIES metamodel is compiled into AP5 types and relations. Transformations operate by making queries and assertions to this database, just as a prototype of a user specification would.

One important advantage of AP5 in connection with this work is that it does not assume any particular internal data structure for storing relations. Programmers can select whichever data structure they see fit. In ARIES, it is convenient to use parse trees as the representation of some of the system description components because they contain program text. In ARIES, the POPART

metaprogramming system is used for this purpose (Wile 1986; Johnson and Yue 1988). However, relations that are not directly part of the program text, such as data flow relations, can also be captured in the same database. The transformations need not be concerned with the particular implementation of the metamodel. Furthermore, an alternate data representation for the ARIES metamodel was also implemented in the REFINE programming environment. This implementation makes it possible for tools in the ARIES system to operate on specifications developed in the KNOWLEDGE-BASED SOFTWARE ASSISTANT Concept Demonstration system, which is primarily written in REFINE (Meyers and Williams 1990).

Transformations as Events

Continuing with the metamodel metaphor, transformations are modeled as events in the metamodel space. A special folder in ARIES called Transformation-Library contains specifications of all the transformations in the system. The same tools for viewing and validating specifications, such as the GIST paraphraser, can be applied to transformations. Likewise, transformations are compiled into Lisp and AP5 functions that operate on the database model of specifications. The explicit specification of transformations enables ARIES to help plan the application of transformations and determine their effects.

Like ordinary events, transformations have input, output, preconditions, goals, and methods. These features are illustrated in figure 4. The inputs are the accessed relation and the accessor as well as the names of the new types and relations that the transformation creates. These input are all typed; the types, such as relation-declaration, are all part of the ARIES metamodel. In general, preconditions and goals are used to determine the applicability and effectiveness of transformations; here, only a goal is defined. The goal in the case of Splice-Data-Accesses is that no information accesses to the spliced relation exist. The method of the transformation is not shown because it involves implementation details that are unlikely to interest an analyst.

Because transformations are specification objects, they can be described using a combination of formal descriptions and hypertext. Some of the textual descriptions shown in figure 4 are extracted from such hypertext descriptions; some are machine-generated natural language.

Effect Descriptions

The principal effects of each transformation are explicitly recorded as part of the transformation definition. Each effect is a generic operation applied to a combination of the transformation's input, output, and other related objects that are not directly input or output. In the case of Splice-Data-Accesses, one splice is performed, and three specification objects are created.

In general, transformations can have two possible effects: main effects and possible effects. *Main effects* are guaranteed to result from a transformation application (assuming that the goal of the transformation is not already satisfied). *Possible effects* might or might not result, depending on the particular situation in which the transformation is applied.

In principle, it should be possible to employ symbolic evaluation tools, such as those of the Specification Assistant, to analyze transformations and automatically determine their effects because the transformations are much like any other events. Currently, however, it is not possible, and the effect descriptions must be recorded separately by the transformation writer.

Retrieving and Using Transformations

When using a library of operators such as our evolution transformation library, a user must do the following: find an appropriate operator, understand what it does, and determine how to apply it to achieve the desired effect.These activities are seldom trivial, particularly if the population of the library is large, and the effects of the operators are potentially complex. Consequently, we have been concerned with providing automated support for these activities. Some new capabilities in these directions were recently developed, and further developments are anticipated.

Effect descriptions are used to assist the transformation retrieval process. The user can specify a desired effect in terms of the class of operation and the objects of interest. For each operand, an object class can be specified, or a particular specification component object can be referred to. Given this description, the retrieval mechanism retrieves three sets of transformations: those that are guaranteed to achieve the desired effect, those that might achieve the desired effect but only in restricted circumstances, and those that achieve part of the desired effect.

The next step is to integrate the retrieval mechanism with the presentation editors being developed for ARIES by Lockheed Sanders. Each presentation is directly manipulable; the user will be able to button on nodes and arcs in diagrams and perform generic operations such as Promote or Splice. The user's gesture will be used as a description of an appropriate transformation to apply. If the gesture unambiguously indicates a particular transformation, the transformation will directly be applied; if not, the user will be asked to disambiguate.

In another application mode, the user will perform a gesture and ask the system to find all transformations that include this gesture as a substep. This request allows the system to suggest macrotransformations that the user might be unfamiliar with. Thus, an ARIES user can start by using the basic 'gestures to edit a system description and gradually move to progressively more powerful and complex transformations.

It is important to note that the mapping between presentation gestures and 'modification operators on the internal representation might be indirect. Figures 4 and 5 show the effects of splicing in the radar track into the air traffic control system from both a presentation standpoint and a representation standpoint. Internally, an accesses-fact relationship holds between Ensure-On-Course and Location-OL Externally, however, the individual accesses-fact relationships are not depicted. Instead, the label on the information flow arc indicates what relations are being accessed. It will be necessary for the presentation system to translate external operations on labels into internal operations on arcs. For beginning users to clearly understand the effects of Splice-Data-Accesses, we demonstrate the transformation on diagrams where each arc has a single label and a single direction. According to the semantics of context diagrams, an arc with multiple labels or directions can be transformed into an equivalent set of arcs with single labels and directions. In this simplified case, the splicing of data accesses directly corresponds to the splicing of information flow arcs.

We have been experimenting with various types of intelligent assistance for the process of applying transformations. In the Specification Assistant, a record was kept of each transformation that was applied and on what parameters. The analyst could undo a sequence of transformations at any time, perform additional changes, and then replay the transformations. The Specification Assistant automatically determined whether the transformations were still applicable and, if not, requested the developer supply new input for the transformations that could not be applied. This mechanism was intended as an aid for supporting the merger of parallel sequences of transformation applications. The ARIES system will build on this capability by using goals and effect descriptions to help suggest transformations to apply and further determine transformation

applicability. It will be possible for the system to take design constraints, such as restrictions on allowed information flow, and automatically suggest transformations that repair constraint violations.

Assessing Library Completeness

The previous analysis gave us the means for determining what transformations to include in our library and the basis for assessing the library's completeness. We distinguish two categories of transformations: basic transformations, which perform some simple operation along one or more dimensions, and macro transformations, which perform multiple operations. We have been extending the basic transformation set to cover all types of operations along all dimensions. Macrotransformations are defined as compound transformations, achieving their effect by invoking a combination of the basic transformations.

Splice-Data-Accesses is an example of such a macrotransformation. Another is Add-Disjoint-Subtypes, first described by Balzer (1985). This transformation defines two disjoint subtypes of a specified type and revises the signatures of all relations over the type so that they are instead restricted to one subtype or the other. The operations of adding the subtypes and specializing the relations are distinct operations, which could be performed independent of each other. Therefore, we realized these individual operations in separate transformations, Add-Specialization and Specialize-Parameter. AddDisjoint-Subtypes now invokes these other transformations as substeps. Users are free to either use the larger command or directly invoke the substeps for some other purpose.

Although the space of possible macrotransformations is unlimited, the space of basic transformations is limited, and we are attempting to achieve complete coverage of the space. At a minimum, we must ensure that each type of link operation can be performed on each type of link and each type of node.

In one sense, this degree of coverage is almost trivially achievable, given suitable general transformations. For example, there is a general Update-Attribute transformation that can update any attribute of any type; this transformation can be used to affect any update operation. There are three reasons why additional work is involved.

First, various types of nodes have necessary conditions associated with them. For example, every concept must have a name and must be part of a folder. Every node type that has distinct conditions should have an associated transformation that can properly create instances of it.

Second, constraints exist among the possible relationships between nodes.
Transformations must respect these constraints. For example, the accessesfact relations on an event must be consistent with the procedural definition of the event. If the definition accesses some piece of information, then an accesses-fact relation must hold between the event and the accessed information. It is not good enough for a transformation to modify the accesses-fact relation by itself; it must also modify the parts of the definition that entail the accesses-fact relation (as is done by Splice-Data-Accesses).

Because our effect descriptions are partial characterizations, there can be more than one possible way to achieve the same effect. For example, there are many ways to split a node, depending on how the attributes of the split node are divided. Thus, it is not enough to require that every kind of node can be split; we must capture each method of node splitting that commonly occurs. We can discriminate among such methods in terms of the links that they affect.

This analysis provides solid guidelines for assessing transformation coverage. Once we provide transformations that perform each possible operation, obeying all constraints, and account for the different possible side effects that can occur, then we can be confident that the library will meet arbitrary user needs. There is still room for variation about how the transformations achieve their required effects, of course. For example, a transformation creating a concept can satisfy the well-formedness constraint that the type be named in several ways (for example, ask the user for a name, or construct a new name at random). We cannot guarantee that our approach to satisfying this constraint (asking the user) will satisfy all users. However, we can expect the basic function of the transformation to agree with user expectations.

Previous work in the development of reusable component libraries has generally been unable to provide methods for assessing library completeness along the lines identified here. For example, Prieto-Diaz and Freeman (1987) factor software component properties along a number of different dimensions but fail to establish a taxonomic hierarchy along any dimension. Thus, there is no way to determine whether one component is more general in applicability than another. Systems that rely on a classification hierarchy of objects (Allen and Lee 1989; see chapter 5) primarily classify on the basis of input and object types. Without a notion of generic operations, the classification of effect is ad hoc at best. By restricting our classification to a particular kind of software component, namely, evolution transformations, we are able to do a much better job of classifying our components.

Applicable Results and Future Challenges

Formalized evolution transformations are a potential benefit to all software-evolution activities, not just specification development. The analysis of transformations in this chapter provides a framework for applying evolution transformations to other languages. Any language that supports the mechanical derivation of semantic relations on software objects is a candidate for formalized evolution. Strongly typed languages such as Ada and Pascal fit in this category. In the case of languages with poor typing mechanisms, annotations introduced by designers can help. For example, Andersen Consulting's Basic Assembler Language Software Reengineering Workbench provides interactive tools for reengineering assembly language programs (chapter 1). Design information is captured through a combination of automatic tools and manual entry. Andersen is considering implementing evolution transformations that operate on such reengineered programs.

This chapter described the semantic basis for developing a reusable library of transformations. Work on extending the library coverage is ongoing. The main technical challenges that remain have to do with providing sufficient automated support for the transformation retrieval and application processes and for deriving effects and preconditions of transformations from their method bodies. Given the work accomplished to date, we believe that it will be straightforward to develop a system that retrieves transformations through the iterative reformulation of queries, as in BACKBORD (Yen, Neches, and DeBellis 1988) and that guides the user in applying the transformation to achieve the desired effect. We envision that the system will ultimately take an active role in the interactive planning of specification changes. Failed preconditions on transformations could then trigger a search of the transformation library for transformations that could make the preconditions true. The system could provide suggestions at each stage about what transformations could be appropriate to perform.

Acknowledgments

We wish to acknowledge the members of the Specification Assistant project . USC/lnformation Sciences Institute for their participation in this research, in particular, Jay Myers, Dan Kogan, Kai Yue. and Kevin Benner. Our colleagues at Lockheed Sanders; David Harris, Jay Runkel, and Paul Lakowski responsible for many of the good ideas behind this work; the bad ideas solely our responsibility. Charles Rich provided useful advice throughout ARIES project. Vairam Alagappan, Van Kelly, Michael Lowry, Jay Myers, d K. Narayanaswamy reviewed earlier drafts of this chapter and made many helpful comments.

References

Abbot, D. A. 1989. KBSA's Requirements Assistant and Aerospace Needs. In Proceedings of Fourth Annual KBSA Conference. Rome, N.Y.: Rome Laboratory.

Air Traffic Operations Service. 1989. 711O.65F: Air Traffic Control. Washington, D.C.: Government Printing Office.

Allen, B. P., and Lee, S. D. 1989. A Knowledge-Based Environment for the Development of Software s Composition Systems. In Proceedings of the Eleventh International Conference on oftware Engineering, 104-112. Washington, D.C.: IEEE Computer Society Press.

Anderson, J. S., and Fickas, S. 1989. A Proposed Perspective Shift: Viewing Specification Design as a Planning Problem. In Proceedings of the Fifth International Workshop on Software Specification and Design, 177-184. Washington, D.C.: IEEE Computer Society Press.

A&A. 1989. *Airman's Information Manual*. Seattle, Wash.: Aviation Supplies and Academics, Inc.

Balzer, R. 1985. Automated Enhancement of Knowledge Representations. In Proceedings of the Ninth International Joint Conference on Artificial Intelligence, 203-207. Menlo Park, Calif.: American Association for Artificial Intelligence.

Balzer, R.: Goldman, N.: and D. S. Wile. 1976. On the Transformational Implementation Approach to Programming. In Proceedings of the Second International Conference on Software Engineering, 337-344. Washington, D.C.: IEEE Computer Society Press.

Balzer, R.; Cohen, D.; Feather, M. S.; Goldman, N. M.; Swartout, W., and Wile, D. S. 1983. Operational Specification as the Basis for Specification Validation. In *Theory and Practice of Software Technology*, eds. D. Ferrari, M. Bolognani, and 1. Goguen. 21-49. Amsterdam: North-Holland.

Bateman, J. 1990. Upper Modeling: Organizing Knowledge for Natural Language Processing. In Proceedings of the Fourth International Natural Language Generation Workshop, 54-61. Pittsburgh. Penn.

Bauer, F. L. 1976. Programming as an Evolutionary Process. In Proceedings of the Second International Conference on Software Engineering, 223-234. Washington, D.C.: lEE Computer Society Press.

Benner, K., and Johnson, W. L. 1989. The Use of Scenarios for the Development and Validation of Specifications. In Proceedings of the Computers in Aerospace VII Conference. New York: AIAA.

Burstall, R. M., and Goguen, J. 1977. Putting Theories Together to Make Specifications. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence, 1045-1058. Menlo Park, Calif.: International Joint Conferences on Artificial Intelligence.

Cohen, D. 1989. AP5 Manual, USC/lnformation Sciences Institute, Marina del Rey, California.

DeBellis, M. 1990. The KBSA Concept Demonstration Prototype. In Proceedings of the Fifth Annual RADC KNOWLEDGE-BASED SOFTWARE ASSISTANT (KBSA) Conference, 211-225. Rome, N.Y.: Rome Laboratory.

Elefante, D. 1989. Overview of the Knowledge-Based Specification Assistant. In Proceedings of the Computers in Aerospace Conference. New York: AIAA.

Feather, M. S. 1990. Specification Evolution and Program (Re)Transformation. In Proceedings of the Fifth Annual RADC KNOWLEDGE-BASED SOFTWARE ASSISTANT (KBSA) Conference, 403-417. Rome, N.Y.: Rome Laboratory.

Feather, M. S. 1989a. Constructing Specifications by Combining Parallel Elaborations. *IEEE Transactions on Software Engineering* 15(2): 198-208.

Feather, M. S. 1989b. Detecting Interference When Merging Specification Evolutions. In Proceedings of the Fifth International Workshop on Software Specification and Design. 169-176. Washington, D.C.: IEEE Computer Society Press.

Fickas, S. 1987. Automating the Specification Process, Technical. Report, ClS-TR-87-05, Dept of Computer and Information Science, Univ. of Oregon.

Fickas, S. 1986. A Knowledge-Based Approach to Specification Acquisition and Construction. Technical Report. 86-1, Dept. of Computer and Information Science, Univ. of Oregon.

Goldman, N. M. 1983. Three Dimensions of Design Development, Technical Report RS-83-2, USC/lnformation Sciences Institute, Marina del Rey, Calif.

Goldman, N.; Wile, D.; Feather, M.; and Johnson, W. L. 1988. GIST Language Description. USC/lnformation Sciences Institute, Marina del Rey, Calif.

Green, c.; Luckham, D.; Balzer, R.; Cheatham, T.; and Rich. C. 1986. Report on a KNOWLEDGE-BASED SOFTWARE ASSISTANT. In *Readings in Artificial Intelligence and Software Engineering.* eds. C. Rich and R. C. Waters, 377-427. San Mateo, Calif.: Morgan Kaufmann.

Grove, P. B., and the Merriam-Webster Editorial Staff. 1971. *Webster's Third New International Dictionary*. Springfield, Mass.: G. & C. Merriam.

Hagelstein, J. 1988. Declarative Approach to Information System Requirements. Journal of Knowledge-Based Systems 1(4): 211-220.

Harris, D. 1988. The Knowledge-Based Requirements Assistant *IEEE Expert* 3(4).·

Huff, K. E., and Lesser, V. R. 1987. The GRAPPLE Plan Formalism, Technical Report, 87-08, Dept. of Computer and Information Science, Univ. of Massachusetts.

Hunt, V., and Zellweger, A. 1987. The FAA's Advanced Automation System: Strategies for Future Air Traffic Control Systems. *IEEE Computer* 20(2): 19-32.

Johnson, P. 1989. Structural Evolution in Exploratory Software Development. In Proceedings of the AAAI Spring Symposium on Software Engineering, 35-39. Menlo Park, Calif.: American Association for Artificial Intelligence.

Johnson, W. L. 1988. Deriving Specifications from Requirements. In Proceedings of the Tenth International Conference on Software Engineering, 428-437. Washington, D.C.: IEEE Computer Society Press.

Johnson, W. L. 1986. Specification via Scenarios and Views. In Proceedings of the Third international Software Process Workshop, 61-63. Washington, D.C.: IEEE Computer Society Press.

The KBSA Project. 1988. Knowledge-based Specification Assistant: Final Report. Marina del Rey, Calif.: USC/Information Sciences Institute.

Johnson W. L., and Yue, K. 1988. An Integrated Specification Development Framework, Technical Report RS-88-215, USC/ Information Sciences Institute, Marina del Rey, Calif.

MacGregor, R. 1989. Loom Users Manual. Marina del Rey, Calif.: USC/ Information Sciences Institute

Myers, J. J., and Williams, G. 1990. Exploiting Metamode1 Correspondences to Provide Paraphrasing Capabilities for the Concept Demonstration. In Proceedings of the Fifth Annual RADC KNOWLEDGE-BASED SOFTWARE ASSISTANT (KBSA) Conference, 331-345. Rome, N.Y.: Rome Laboratory.

Narayanaswamy, K. 1988. Static Analysis-Based Program Evolution Support in the Common Framework, 222-230. In Proceedings of the Tenth International Software Engineering Conference. Washington. D.C.: IEEE Computer Society Press.

Niskier, C.; Maibaum. T.; and Schwabe, D. 1989. A Look through PRISMA: Towards Pluralistic Knowledge-Based Environments for Software Specification Acquisition. In Proceedings of the Fifth International Workshop on Software Specification and Design, 128-136. Washington, D.C.: IEEE Computer Society Press.

Prieto-Diaz, R.. and Freeman, P. 1987. Classifying Software for Reusability. *IEEE Software* 5(1):6-16.

Reasoning Systems. 1986. REFINE User's Guide. Reasoning Systems. Palo Alto. Calif.

Reubenstein, H. B., and Waters. R. C. 1989. The Requirements Apprentice: An Initial Scenario. Proceedings of the Fifth International Workshop on Software Specification and Design, 211-218. Washington, D.C.: IEEE Computer Society Press.

Rich, C.; Schrobe. H. E.; and Waters. R. C. 1979. An Overview of the PROGRAMMER'S APPRENTICE. In Proceedings of the Sixth International Joint Conference on Artificial Intelligence, 827-828. Menlo Park. Calif.: International Joint Conferences on Artificial Intelligence.

Robinson, W. N. 1989. Integrating Multiple Specifications Using Domain Goals. In Proceedings of the Fifth International Workshop on Software Specification and Design. 219-226. Washington D.C.: IEEE Computer Society Press.

Swartout. W. 1982. GIST English Generator. In Proceedings of the First National Conference on Artificial Intelligence. 404-409. Menlo Park. Calif.: American Association for Artificial Intelligence.

Waters, R. C. 1985. The Programmer's Apprentice: A Session with KBEmacs. *IEEE Transactions on Software Engineering* SE-11(11): 1296-1320.

Wile, D. S. 1983. Program Developments: Formal Explanations of Implementations. In *New Paradigms for Software Development*, ed. W. Agresti, 239-248. Washington, D.C.: IEEE Computer Society Press.

Yen. J.; Neches, R.; and DeBellis. M. 1988. Specification by Reformulation: A Paradigm for Building Integrated User Support Environments. In Proceedings of the Seventh National Conference on Artificial Intelligence, 814-819. Menlo Park, Calif.: American Association for Artificial Intelligence.

Yue, K. 1989. Representing First Order Logic-Based Specifications in Petri-Net-Like graphs. In Proceedings of the Fifth International Workshop on Software Specification and Design, 291-293. Washington. D.C.: IEEE Computer Society Press.